UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO
INF01151 – SISTEMAS OPERACIONAIS II N

Bruno Marques Bastos - 00314518
Eduardo Raupp Peretto - 00313439
Marcel Ramos do Carmo - 00314937
Vinícius Matte Medeiros - 00330087

Relatório Trabalho - Parte 1

Porto Alegre
2023

Sumário

Descrição dos ambientes de teste

Sistema operacional e distribuição:

Ubuntu 22.04.3 LTS (64 bits)

Configuração da máquina:

Processador: Intel® Core™ i7-8550U CPU @ 1.80GHz × 8

Memória: 8GB

Compiladores utilizados:

gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0

Sistema operacional e distribuição:

VirtualBox executando Ubuntu 22.04.3 LTS

Configuração da máquina:

Processador: Ryzen 7 4800H

Memória: 16GB

Sendo 4 núcleos de processamento e 8GB de memória alocados para o virtual box.

Compiladores utilizados:

gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0

Sistema operacional e distribuição:

MacOS 14.1.1

Configuração da máquina:

Processador: M1 Pro (ARM64)

Memória: 16GB

Compiladores utilizados:

Apple clang version 14.0.3 | Target: arm64-apple-darwin23.1.0

Sistema operacional e distribuição:

VirtualBox executando Ubuntu 22.04.3 LTS

Configuração da máquina:

Processador: Intel Core i5 10400 @ 2.90GHz

Memória: 16GB

Sendo 2 núcleos de processamento e 4GB de memória alocados para o virtual box.

Compiladores utilizados:

gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0

Justificativas

(A) Como foi implementada a concorrência no servidor para atender múltiplos clientes

A implementação de acesso de múltiplos clientes foi implementada utilizando uma estrutura atômica de lista encadeada, essa estrutura é identificada pelo <*username*> do cliente e ela é capaz de armazenar o número de conexões que um dado cliente possui, sendo necessário um vetor de duas posições para armazenar os descritores dos sockets para possíveis verificações, como pedido de desconexão. Inicialmente a lista é inicializada vazia e cada vez que um cliente solicitar uma conexão é verificada a existência do <*username*> nessa lista e a quantidade de conexões disponíveis. Dada a possibilidade de concorrência entre vários clientes conectando-se ao servidor, também surge a chance de inserir dois clientes distintos no mesmo espaço da lista de clientes conectados. Isso pode resultar na sobreposição de informações de usuários. Para evitar esse cenário, foi implementado um mutex que permite apenas a inserção sequencial de um cliente por vez na lista de clientes ativos do servidor.

(B) Em quais áreas do código foi necessário garantir sincronização no acesso a dados

Foi necessário garantir a sincronização no acesso a dados, tanto no lado cliente como no lado servidor utilizando a primitiva de sincronização *inotify* do *unix*. No lado cliente, operações aplicadas diretamente na pasta *sync_dir_<username>*, quando algum arquivo é movido, alterado, modificado, criado, ou excluído, é enviado um *signal* para os outros devices conectados e relacionados com o *user* atual, do mesmo modo quando ocorre alterações por comandos *upload* ou *delete*. Já no lado servidor, quando é percebido um *signal* de alteração, o servidor é responsável por propagar as modificações realizadas por um dos devices para os demais, mantendo assim uma sincronização entre os diferentes dispositivos.

No arquivo server.c, foram empregados mutex para realizar a inclusão e exclusão de conexões com mesmo número de usuário, de forma a se evitar condição de corrida na variável que limita o número de conexões por nome de usuário ao mesmo tempo.

(C) Descrição das principais estruturas e funções que você implementou

As funções e estruturas utilizados no trabalho foram subdivididas em três partes, as que são utilizadas somente pelo cliente, as que são utilizadas somente pelo servidor, e as que são utilizadas por ambas. Ainda, dentro das funções utilizadas pelo cliente, foi criado um arquivo separadamente que realiza somente a interação com o usuário (interface). Segue abaixo tabela contendo explicações dos usos das funções e estruturas utilizadas.

Cliente - Funções	
Nome	**Descrição**
printUsage	**Entrada**: (void) **Saída**: (void) Imprime uma mensagem com dicas de como o usuário deve fornecer as entradas para estabelecer conexão.

getServerHost	**Entrada**: (char *) nome_do_host **Saída**: (struct *hostent) estrutura_com_informacoes_do_host Usada para obter informações de um host a partir do seu nome.
createSocket	**Entrada**: (void) **Saída**: (int) descritor_do_socket Cria um socket do tipo TCP.
inicializeServerAddress	**Entrada**: (struct *hostent) informacoes_do_host, (int) porta **Saída**: (struct sockaddr_in) informacoes_do_servidor Inicializa uma estrutura que irá representar o endereço do servidor.
connectToServer	**Entrada**: (int) socket(struct sockaddr_in) infos_do_servidor **Saída**: (void) Realiza a tentativa de se conectar com o servidor através do socket especificado.
check_login_response	**Entrada**: (int) descritor_do_socket **Saída**: (int) resposta_do_servidor Verifica o retorno do servidor após a tentativa de conexão. Se obtiver sucesso é retornado 0, caso contrário é retornado -1.
start_inotify	**Entrada**: (void *) descritor_do_socket **Saída**: (void) Inicializa o monitoramento de um dado diretório específico, trata eventos relevantes (como modificações de arquivos) e envia notificações para o cliente por meio do socket.
handle_inotify_event	**Entrada**: (int) descritor_de_arquivo, (int) descritor_do_socket, (char*) caminho, (char*) nome_usuario **Saída**: (void) Processa eventos inotify relevantes, exibindo mensagens e chamando funções específicas para manipular os arquivos no servidor, como deletar ou transferir.
get_client_file_array	**Entrada**: (const char*) caminho_base, (char [][256]), (int*) contador_de_arquivo **Saída**: (void) Obtém a lista de arquivos do cliente em um determinado diretório e utiliza o ponteiro para saber a quantidade de arquivos em uma determinada pasta.
handleInitialSync	**Entrada**: (void*) argumentos_da_thread_ptr **Saída**: (void*) Função responsável por lidar com as sincronização inicial dos arquivos entre o cliente e o servidor. É chamada pela get_sync_dir.
delete_local_file	**Entrada**: (const char*) nome_arquivo, (const char*) nome_usuario, (const char*) caminho_arquivo **Saída**: (void) Exclui um arquivo local do sync_dir do cliente.

watch_server_changes	**Entrada**: (void*) argumentos_thread **Saída**: (void *) Função responsável por observar as mudanças no servidor e realiza a sincronização necessária.
upload_file	**Entrada**: (const char*) caminho_local_do_arquivo, (int) socket **Saída**: (int) status_do_upload Encapsula a lógica para enviar um arquivo para o servidor, incluindo o envio do nome do arquivo e dos dados do arquivo, utilizando pacotes e funções auxiliares (create_packet, send_packet_to_socket, destroy_packet, get_file_size, read_file_into_buffer). O status do upload é 0 caso haja sucesso na operação, caso contrário é -1.
download_file	**Entrada**: (const char*) nome_do_arquivo, (int) socket, (int) em_sync_dir, (const char*) nome_usuario **Saída**: (int) status_do_download Essa função encapsula a lógica necessária para baixar um arquivo do servidor para o cliente, gerenciando a comunicação entre o cliente e o servidor e a gravação do conteúdo do arquivo localmente. Funções auxiliares: (create_packet, send_packet_to_socket, receive_packet_from_socket). O status do download é 0 caso haja sucesso na operação, caso contrário é -1.
delete_file	**Entrada**: (const char*) nome_do_arquivo, (int) socket, (cost char*) nome_usuario **Saída**: (int) status_do_delete Essa função encapsula a lógica necessária para enviar uma solicitação de exclusão de arquivo para o servidor. A efetiva exclusão do arquivo no servidor seria tratada pelo lado do servidor, que interpreta a solicitação e realiza as operações necessárias no sistema de arquivos do servidor. Funções auxiliares: (create_packet, send_packet_to_socket). O status do delete é 0 caso haja sucesso na operação, caso contrário é -1.
list_server	**Entrada**: (int) socket **Saída**: (int) status_do_list_server Fornece uma maneira simples para os clientes obterem e visualizarem a lista de arquivos disponíveis no servidor. Funções auxiliares: (create_packet, send_packet_to_socket, receive_packet_from_socket). O status do list_server é 0 caso haja sucesso na operação, caso contrário é -1.
list_client	**Entrada**: (int) socket, (const char *) usuario **Saída**: (int) status_do_list_client Fornece uma maneira para os clientes exibirem a lista de get_sync_dirarquivos disponíveis localmente. Funções auxiliares: (get_file_metadata_list). O status do list_client é 0 caso haja sucesso na operação, caso contrário é -1.
close_connection	**Entrada**: (int) socket **Saída**: (int) status_do_exit

	Facilita o processo de encerramento da conexão com o servidor. Ela envia um comando de saída, aguarda a confirmação do servidor e retorna o status correspondente.
get_sync_dir	**Entrada**: (const char *) nome_usuario, (int) socket **Saída**: (int) status_do_get_sync_dir Verifica se existe uma pasta correspondente ao usuário no servidor. Se a pasta existir, a função faz o download de todos os arquivos associados a essa pasta. Se a pasta não existir, a função cria uma nova pasta vazia no servidor. Em resumo, a função sincroniza os diretórios entre cliente e servidor, garantindo que os arquivos do usuário estejam sempre atualizados em ambos os lados. O status do get_sync_dir é 0 caso haja sucesso na operação, caso contrário é -1.
receive_data	**Entrada**: (int) socket, (void*) buffer_de_armazenamento, (size_t) numero_de_bytes_a_receber, (int) tempo_de_espera **Saída**: (int) bytes_recebidos Projetada para receber dados de um soquete, garantindo que uma quantidade específica de dados seja lida do socket. O retorno da função é -1 quando atingi o tempo de espera estipulado ou há algum erro de leitura.
parse_input	**Entrada**: (char*) input_de_dados, (int) socket **Saída**: (int) status_do_parser Interpreta os comandos fornecidos pelo usuário, realizando a ação correspondente no servidor e lidando com possíveis erros. O status do parser retorna 0 quando ocorreu algum erro e 1 caso a conexão foi encerrada, que é usada para o controle de laço usado no interface do usuário.
printOptionsMenu	**Entrada**: (void) **Saída**: (void) Mostra ao usuário quais as opções de comando o sistema espera que ele digite e solicita uma entrada.
userInterface	**Entrada**: (void*) ponteiro_para_o_socket **Saída**: (void) Cria uma interface simples para o usuário interagir com o programa, fornecendo um menu de opções e processando os comandos fornecidos pelo usuário até que ele decida sair.

A função principal do lado cliente realiza todas as operações relacionadas ao estabelecimento de conexão com o servidor e imprime uma interface separadamente em uma *thread* para os usuários entenderem quais funções são oferecidas pelo sistema. Os pacotes de dados criados são enviados para que o servidor possa tratar e realizar as funções de maneira a dar um retorno a cada cliente ativo.

Servidor - Estruturas	
Nome	**Atributos**

list_users_t	(char*) username - Nome do usuário. (int) connections - Quantidade de conexões ativas. (int) socket[2] - Identificação dos sockets. (list_users_t*) next - Próximo da lista encadeada.
notify_data_t	(char*) username; (int) socket;

Servidor - Funções	
Nome	**Descrição**
create_new_user	**Entrada**: (char *) nome_do_cliente, (int) socket **Saída**: (list_users_t*) lista_de_usuarios_atualizada Responsável por criar um nodo na lista de usuário e inicializá-lo com os parâmetros passados.
send_connection_response	**Entrada**: (int) resposta, (int) socket **Saída**: (int) status_connection_response Envia o status da conexão como resposta para o client.
insert_or_update_new_connection	**Entrada**: (list_users_t*) lista_de_usuarios, (char*) nome_do_usuario, (int) socket, (int*) status_da_conexao **Saída**: (list_users_t*) lista_de_usuarios_atualizada Responsável por verificar a existência do usuário na lista de usuários e inserir novos clientes ou atualizar caso já exista alguma conexão ativa, não permitindo mais de 2 conexões simultâneas por usuários.
remove_user_connection	**Entrada**: (list_users_t*) lista_de_usuarios, (char*) nome_do_usuario, (int) socket **Saída**: (list_users_t*) lista_de_usuarios_atualizada Responsável por realizar a remoção de um cliente específico da lista de clientes com base no socket.
print_user_list	**Entrada**: (list_users_t*) lista_de_usuarios **Saída**: (void) Usada para debbug sobre a lista de usuários.
free_user_list	**Entrada**: (list_users_t*) lista_de_usuarios **Saída**: (void) Responsável por liberar a memória alocada para a lista de usuários de maneira recursiva.
setupSocket	**Entrada**: (int*) socket_ptr, (int) porta **Saída**: (int) status_do_setup Responsável por configurar e preparar um socket no lado do servidor para aceitar conexões de clientes. Retorna 0 caso haja sucesso e -1 caso algum erro.
handle_packet	**Entrada**: (thread_data_t*) dados_da_thread, (int*) status_da_conexao

	Saída: (int) status_do_processamento Responsável por processar os pacotes recebidos pelo servidor. A função identifica o tipo de comando contido no pacote e chama funções específicas para lidar com cada tipo de comando. Esses comandos podem incluir operações como login, upload, download, exclusões, listagem de arquivos no servidor e no cliente, obtenção da pasta de sincronização no cliente e encerramento da conexão. Além disso, a função também pode modificar o estado da conexão, indicando se ela deve ser encerrada ou não.
get_socket_notify	**Entrada**: (const char*) nome_usuario, (int [2]) resultado **Saída**: (void) Permite que o servidor envie notificações direcionadas aos sockets quando ocorrerem mudanças relevantes no sistema.
send_changes_to_clients	**Entrada:** (char *) nome_usuario, (type_packet_t)tipo_pacote, (char*) nome_arquivo, (int) socket **Saída:** (void) Cria um pacote com as mudanças e as envia para os devidos clientes.
update_socket_notify	**Entrada:** (const char*) nome_usuario, (int) socket **Saída:** (void) Atualiza o *obserser* do notify para as mudanças realizadas nas pastas
create_folder	**Entrada**: (char) nome_do_usuario* **Saída**: (void) Responsável por criar uma pasta para armazenar os arquivos sincronizados de um usuário específico. O caminho da pasta é construído concatenando o caminho base (SYNC_DIR_BASE_PATH) com o nome de usuário. O código utiliza a função mkdir para criar o diretório no sistema de arquivos.
handle_new_client_connection	**Entrada**: (void*) argumentos_passados_a_thread **Saída**: (void*) Responsável por gerenciar a conexão com um novo cliente no servidor, recebendo e processando pacotes do cliente, atualiza a lista de clientes conectados, se necessário.
send_file	**Entrada**:(int) socket_cliente, (char*) nome_do_arquivo_a_ser_enviado, (char*) caminho_do_diretorio **Saída**: (int) status_do_send Responsável por enviar um arquivo para um cliente conectado ao servidor. Ela constrói o caminho

	completo do arquivo, abre o arquivo no modo binário para leitura, cria um pacote contendo o nome do arquivo como payload, aloca dinamicamente um buffer para o conteúdo do arquivo, lê o arquivo para o buffer do payload, e envia o pacote com o conteúdo do arquivo para o cliente através do socket, fechando o arquivo em seguida. Em caso de sucesso, imprime uma mensagem indicando que o arquivo foi enviado com êxito = 0, caso falhe -1.
receive_file	**Entrada**: (int) socket_cliente, (const char*) nome_do_usuario, (const char*) nome_do_arquivo, (uint32_t) tamanho_payload **Saída**: (int) status_do_receive Responsável por receber um arquivo do cliente conectado ao servidor. Ela inicia recebendo um pacote contendo os dados do arquivo do socket do cliente. Em seguida, verifica se o pacote é válido, contém dados do tipo DATA e tem um tamanho de payload maior que zero. A função aloca dinamicamente memória para o nome do arquivo, constrói o caminho do arquivo utilizando o diretório do usuário e o nome do arquivo, abre o arquivo no modo de escrita binária e escreve o payload do pacote no arquivo. Em caso de sucesso, imprime uma mensagem indicando que o arquivo foi recebido e salvo com êxito = 0, caso falhe retorna -1.
delete_file	**Entrada**: (int) socket_cliente, (const char*) nome_arquivo_ser_excluido, (const char*) localizacao_diretorio **Saída**: (int) status_delete Responsável por excluir um arquivo no lado do servidor. Ela constrói o caminho completo do arquivo usando o nome do diretório (filepath) e o nome do arquivo (filename). Em seguida, utiliza a função remove para excluir o arquivo. Se a operação for bem-sucedida, a função imprime uma mensagem indicando que o arquivo foi excluído com sucesso e retorna 1 (status_delete = 1). Em caso de falha, a função imprime uma mensagem de erro, retorna -1 e indica que houve um erro ao excluir o arquivo.
list_server	**Entrada**: (int) socket_cliente, (const char*) caminho_do_diretorio_do_usuario **Saída**: (int) status_list_server Responsável por listar os arquivos disponíveis no servidor para um cliente específico. Utiliza a função get_file_metadata_list para obter metadados dos arquivos no diretório especificado por userpath. Os metadados são concatenados em uma string file_list. Em seguida, é criado um pacote (packetFileList) do tipo CMD_LIST_SERVER contendo a lista de

	arquivos como payload. A função envia esse pacote para o cliente usando o socket (client_socket). Se a operação for bem-sucedida, a função retorna 0 (status_list_server = 0). Em caso de falha, imprime uma mensagem de erro, destrói o pacote e retorna -1, indicando que ocorreu um erro ao enviar a lista de arquivos para o cliente.

Após a inicialização do servidor, o mesmo aguarda a conexão de um novo cliente, a cada novo cliente o servidor cria uma *thread* para unitária para cada device de cliente que tentar realizar uma nova conexão, enviando mensagens de erro ao cliente que tentar realizar mais de duas conexões. Através dos pacotes enviados pelos clientes, o servidor é capaz de produzir os comandos enviados a partir de um cliente.

Ambas (Commons.h) - Estruturas	
Nome	**Atributos**

type_packet_t	enum {
	DATA,
	CMD_LOGIN,
	CMD_UPLOAD,
	CMD_DOWNLOAD,
	CMD_DELETE,
	CMD_LIST_SERVER,
	CMD_LIST_CLIENT,
	CMD_GET_SYNC_DIR,
	CMD_WATCH_CHANGES,
	CMD_NOTIFY_CHANGES,
	CMD_EXIT,
	INITIAL_SYNC,
	FINISH_INITIAL_SYNC,
	FILE_LIST
	}

packet_t	(type_packet_t) type - Tipo do pacote (uint32_t) length_payload - Tamano do payload (char*) payload - Conteúdo do pacote
thread_data_t	(struct sockaddr_in) serv_addr; (packet_t) packet; (char*) userpath; (char*) username; (int) socket;
(struct) ThreadArgs	(const char*) username; (int) socket;

Ambas (Commons.h) - Funções

Nome	Descrição
create_packet	**Entrada**: (type_packet_t) tipo_do_pacote, (const char*) conteudo_do_pacote, (int) tamanho_do_payload **Saída**: (packet_t*) pacote_criado Função responsável por criar e inicializar um pacote dados os parâmetros de entrada. Esse pacote pode ser enviado tanto pelo cliente quanto pelo servidor.
destroy_packet	**Entrada**: (packet_t*) pacote_a_ser_destruido **Saída**: (void) Função responsável por destruir um pacote em caso seja necessário, liberando a memória alocada.
get_packet_type_name	**Entrada**: (type_packet_t) tipo_do_pacote_alvo **Saída**: (const char*) nome_do_tipo_do_pacote_alvo Função usada para debugar o tipo de pacote.
print_packet	**Entrada**: (packet_t*) pacote_alvo **Saída**: (void) Função usada para debug. Imprime os atributos do pacote
send_packet_to_socket	**Entrada**: (int) socket, (const packet_t*) pacote_alvo_envio **Saída**: (int) status_do_send Função responsável por enviar pacotes entre cliente e servidor. Retorna 0 caso o pacote tenha sido enviado com sucesso e -1 quando algum erro é detectado.
receive_packet_from_socket	**Entrada**: (int) socket **Saída**: (packet_t*) pacote Função responsável pelo recebimento de pacotes trocados entre cliente e servidor, caso haja algum erro o pacote é destruído e retornado NULL.
receive_packet_wo_payload	**Entrada**: (int) socket **Saída**: (packet_t*) pacote Função responsável por receber os pacotes trocados entre cliente e servidor, fazendo os possíveis tratamento de erros e retornando o pacote recebido ou NULL caso dê erro.
receive_packet_payload	**Entrada**: (int) socket, (packet_t*) pacote_alvo **Saída**: (int) status_desempacotamento Função responsável por fazer o desempacotamento do conteúdo do pacote, retornando -1 caso ocorra algum erro na leitura dos dados recebidos.
clone_string	**Entrada**: (const char*) string_source **Saída**: (char*) string_destination A função faz uma cópia em alocação dinâmica da string passada de argumento e retorna o endereço da cópia.
is_equal	**Entrada**: (const char*) string1, (const char*) string2 **Saída**: (int) status_comparacao A função retorna 1 se as strings passadas como

	argumento forem iguais, e retorna 0 se elas forem diferentes.
print_socket_info	**Entrada**: (struct sockaddr_in) cli_addr **Saída**: (void) A função imprime o endereço da conexão feita pelo socket.
get_file_metadata_list	**Entrada**: (const char*) basepath, (char*) lista_de_arquivos **Saída**: (void) A função lê os arquivos no diretório no caminho passado como argumento e armazena os metadados do arquivo no endereço apontado por file_list.
get_file_size	**Entrada**: (const char*) nome_do_arquivo **Saída**: (long) tamanho_do_arquivo A função retorna o tamanho do arquivo passado como argumento.
read_file_into_buffer	**Entrada**: (const char*) nome_do_arquivo **Saída**: (char*) conteudo_arquivo A função aloca um buffer e o preenche com a leitura do arquivo passado como argumento.
create_folder	**Entrada**: (char) username[50] **Saída**: (void) A função cria uma pasta sync_dir_<username> se já não existir uma pasta com esse nome.

Nessa parte são implementadas as funções utilizadas tanto pelo lado cliente como no lado servidor, salvos algumas estruturas que possuem atributos que são usados por um mas não por outro, um exemplo disso é o que acontece com a estrutura *thread_data_t.*

As estruturas criadas foram escolhidas para simplificar e facilitar a manipulação das primitivas de comunicação e sincronização, porém como a implementação foi feita totalmente utilizando a linguagem C, tivemos que tomar cuidado ao utilizar ponteiros para evitar vazamentos de memória.

(D) Explicar o uso das diferentes primitivas de comunicação

Em vista as necessidades do projeto, foi observado que seria necessário somente o uso de mutexes, uma vez que sua implementação é mais simples e intuitiva para garantir a exclusão mútua em alguns pontos específicos do código para o controle de acesso a recursos que são importantes não serem alterados simultaneamente. Poderíamos ter implementado com semáforos, porém preferimos aderir ao método mais simples dadas às dificuldades enfrentadas durante a implementação de outros trechos do código.

Descrição dos problemas

Os principais problemas e dificuldades encontradas pelo grupo foram questões relacionadas à primitivas de sincronização, uma vez foram observadas dúvidas referentes às diferentes heurísticas possíveis para o tratamento e complexidade intrínseca de cada

uma delas. Primeiramente pensamos de uma maneira simples, onde ao aplicar modificações tanto por força bruta (alterações realizadas diretamente na pasta de arquivos sincronizados) quanto utilizando operações de input do sistema, o monitor faz com que o cliente realize operações forçadas de envios de pacote de upload, download e delete ao servidor, fazendo que o mesmo receba os arquivos que não possuía ao mesmo tempo em que envia ao cliente os arquivos faltantes.

Após o upload de mais de um arquivo, separadamente, notou-se perda de informação e problemas na escrita dos arquivo enviados no servidor após o primeiro, devido a uma declaração equivocada do tamanho de um buffer que não considerava o caractere '\0' no final das strings de nome de arquivo. Isso afetou o envio de payloads importantes.

Na definição dos "ifs" da função handle_inotify_event, foi considerado utilizar a flag IN_CLOSE_NOWRITE para detectar mudança no a_time de arquivos, porém essa detecção foi descartada pois ao abrir (somente) um arquivo, o watch do inotify retornava um sinal para todos os arquivos do diretório, que não era o comportamento desejado.

Uma das dificuldades foi que ao utilizar C para o desenvolvimento da aplicação, tivemos que tomar muito cuidado com a manipulação de ponteiros e a liberação de memória. O grupo acabou decidindo que C seria a linguagem apropriada pois todos os membros tinham familiaridade devido ao aprendizado passado de outras disciplinas, embora saibamos que a implementação com a linguagem C++ seria mais fácil de realizar, nenhum dos integrantes tinha conhecimento sobre esta última, sendo necessário, talvez, o emprego de mais tempo para entender as demais funcionalidades oferecidas.